The Patent Office
PATENTS • DESIGNS • COPYRIGHT • TRADE MARKS
INVESTOR IN PEOPLE

Application No: GB 9928341.8
Claims searched: 1-8
Examiner: Paul Jefferies
Date of search: 6 November 2001

Patents Act 1977
Search Report under Section 17

Databases searched:

UK Patent Office collections, including GB, EP, WO & US patent specifications, in:

UK Cl (Ed.S): G4A (APL)

Int Cl (Ed.7): G06F 9/45

Other: ONLINE: WPI, EPODOC, JAPIO, INSPEC

Documents considered to be relevant:

Category	Identity of document and relevant passage	Relevant to claims
X	GB 2127188 A (TEKTRONIX) See page 1, lines 45-50; page 2, lines 40-45; page 4, line 35 et seq.; page 6 lines 4-25 and figure 1.	1, 4, 5, 7
X	INSPEC Abstract Accession No. 3029114 & "IEEE 1987 National Aerospace and Electronics Conference (NAECON)" published 1987, IEEE, pages 728-731, Vol 3, R A Lawler "Automatic translation of assembly language software" (see abstract).	1 at least

X	Document indicating lack of novelty or inventive step	A	Document indicating technological background and/or state of the art.
Y	Document indicating lack of inventive step if combined with one or more other documents of same category.	P	Document published on or after the declared priority date but before the filing date of this invention.
&	Member of the same patent family	E	Patent document published on or after, but with priority date earlier than, the filing date of this application.

(12) UK Patent Application (19) GB (11) 2 127 188 A

(21) Application No 8315523
(22) Date of filing
7 Jun 1983
(30) Priority data
(31) 387729
(32) 14 Jun 1982
(33) United States of America (US)
(43) Application published
4 Apr 1984
(51) INT CL³ G06F 15/20
(52) Domestic classification
G4A PX
(56) Documents cited
GB 1372430
EP A 0049176
US 4309756
US 3938103
(58) Field of search
G4A
(71) Applicant
Tektronix Inc
(USA–Oregon)
4900 SW Griffith Drive
PO Box 500
Beaverton
Oregon 97077
United States of America
(72) Inventor
Henry Jonge Vos
(74) Agent and/or Address for Service
Langner Parry
52/54 High Holborn
London WC1V 6RR

(54) Software/hardware integration control system

(57) The system (ICS) provides an automatic method for formatting a machine independent program written in a high level language to run on a prototype processor system without a detailed knowledge of the assembler, the linker and the language interface requirements.

ICS provides a list of the items you need to specify in order to configure a program to a prototype processor system—everything from the name of your compiled high level language object program to the address where the program begins execution. Based on the entries, ICS generates the configuration object file and linker commands needed to configure your prototype processor system.

Because you use ICS's high-level language directives, you can describe the prototype processor system more quickly and with fewer errors. And ICS checks the validity of your statements, thus saving you from errors that would not be caught until later.

Finally, the ICS generated configuration file and linker commands are combined with the high level language object program and the language run-time support library to generate an executable load module. This module may then in turn be utilized to burn the necessary PROMs for the prototype processor system.

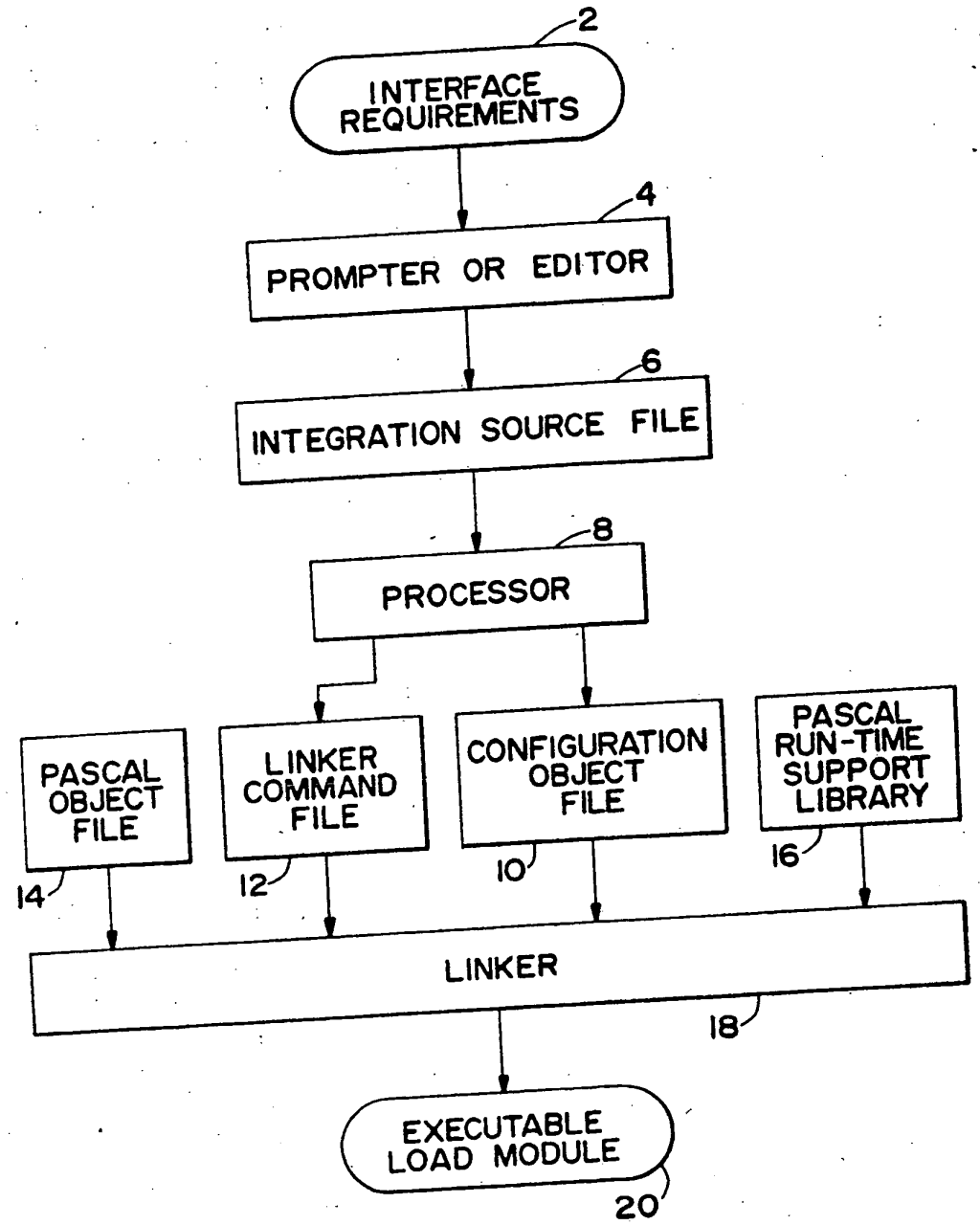
2
INTERFACE REQUIREMENTS
4
PROMPTER OR EDITOR
6
INTEGRATION SOURCE FILE
8
PROCESSOR
PASCAL OBJECT FILE
14
LINKER COMMAND FILE
12
CONFIGURATION OBJECT FILE
10
PASCAL RUN-TIME SUPPORT LIBRARY
16
LINKER
18
EXECUTABLE LOAD MODULE
20

Fig 1.

2127188

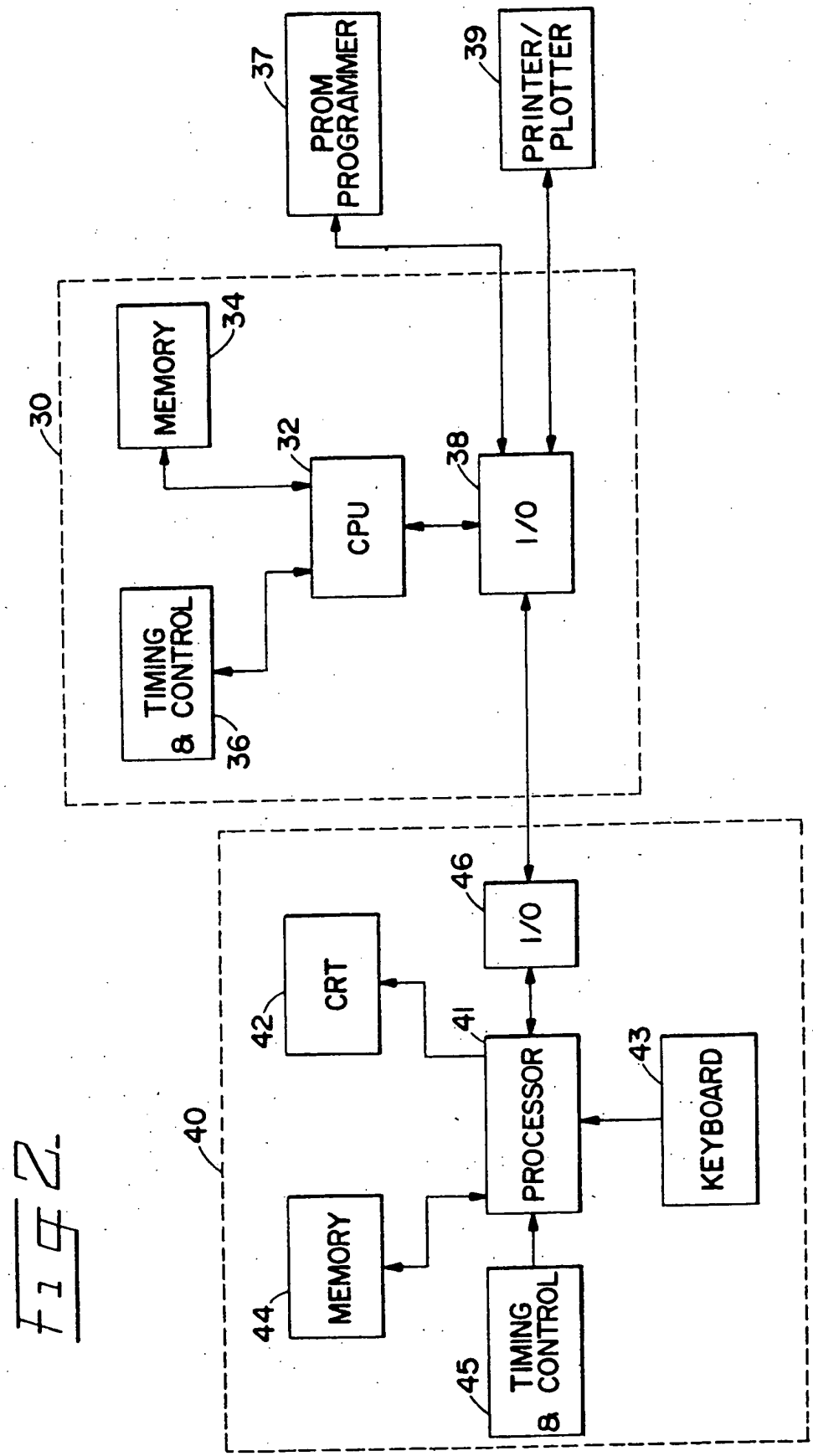

FIG 2

SPECIFICATION

Software/hardware integration control system

Background of the Invention

This invention relates to the interfacing of software to hardware, more specifically to a system for translating hardware/software interface specifications simultaneously into specific microprocessor executable code and commands for the linker/loading system of a selected hardware configuration.

Computer based instruments are in fact systems of mechanical and electronic components interacting with the computer program stored in those hardware components. The task of correctly interfacing hardware and software has always been a rather intricate one and very time consuming. The system of the present invention allows the instrument designer to specify the hardware/software interface in a high order language in an interactive way. That system then translates those specifications into code executable by the instrument computer as well as other commands to be executed by the program linking/loading systems. It thus reduces to minutes a development proces that might otherwise take up to several days or even weeks.

When a high order language, such as Pascal is used, the program is machine independent by virtue of the nature of the language. Pascal source programs do not vary, regardless of the process or host computer on which it is to be used. However, with conventional Pascal, there's no direct way to specify implementation-dependent requirements such as interrupt vectors, restart routines, or memory configuration.

You could develop a large assembly language routine (to connect your Pascal program with the prototype hardware) and a linker command file (to specify your memory configuration). To do this, you would need detailed knowledge of the assembler, the linker, and the Pascal interface requirements. The task is time-consuming, and with so many low-level details to keep track of, errors are inevitable. There are no known prior art systems for generating the linker commands and the configuration object files automatically.

It would be desirable to have a system which provides you with a list of the items you need to specify in order to configure a program to a protoype—everything from the name of your compiled object program to the address where the program begins execution. Then, based on your reponses, generates the configuration object file and linker commands needed to configure your prototype.

If such a system used high-level language directives, you could describe your prototype more quickly and with fewer errors. And the system could also check the validity of your statements, thus saving you from errors that would not be caught until later. It is believed that the present invention embodies such a system.

Summary of the Invention

In accordance with the illustrated embodiment, the disclosed integration control system (ICS) relieves the computer based instrument designer from having to design and debug hundreds to thousands of lines of computer level code as well as having to familiarize himself with increasingly complex linker/loading systems requiring dozens of specific commands in a special linker command language.

The present invention provides a method and a system for integrating a high level language program together with the hardware limitations of a selected prototype processor. This is accomplished automatically by interacting with the designer to prepare a source file which includes software, hardware and interrupt configuration specifications for a selected prototype processor. It also includes processing of the source file generated above to generate a linker command file and a configuration object file.

The linker command file controls the linking process of the high level language program with the configuration object file to generate a load module executable by the prototype processor. During linking selected routines from the run-time library for the particular high level language may also be included as necessary.

The interaction with the designer includes the promoting of the designer for necessary inputs as to software and prototype processor hardware and interrupt specifications, then, in response to those inputs a source file is created.

The configuration object file referred to above includes interrupt vectors, interrupt service routines, a reset routine and a program initialization routine for the prototype processor.

Brief Description of the Drawings

Figure 1 shows a flow diagram of the integration control system of the present invention.

Figure 2 is a simplified block diagram of a terminal interconnected with a host computer representing the embodiment of the present invention.

Description of the Preferred Embodiment

When a high order language, such as Pascal, is utilized to generate a machine independent program the code produced by the compiler cannot be executed directly. The proper codes in the run-time library must be linked with the compiled code. Depending on the hardware and the software needs of the prototype processor system, the prototype processor system must be initialized for the program. The linker must be told where to store the generated code in the memory of the prototype processor system.

The software/hardware integration control system of the present invention functions generally as shown in the flow chart of Fig. 1. Block 2 represents the designers interactive specifications of the hardware, software and interrupt configurations of the prototype in response to prompts from the prompter of block 4. Those specifications are then utilized by prompter 4 to generate the integration source file of block 6. The source file of block 6 is processed by the processor (block 8) producing the linker command file (block 12) and the configuration object file (block 10).

The Pascal object file (block 14), the software to be run on the prototype processor system, together with the Pascal run-time support library (block 16), the configuration object file (block 10), and the linker command file (block 14) are applied to linker 18. In linker 18, the configuration object file is linked to the Pascal object files and the run-time support library. Under the control of the linker command file, linker 18 will produce a load file (block 20) which can be executed by the prototype processor system specified in response to prompter 4.

The configuration object file (block 10) includes interrupt vectors, interrupt service routines, a reset routine and program initialization routine for the prototype processor system specified. The linker command file (block 12) generated for the particular prototype processor system ties together the Pascal object code (block 14), the configuration object file (block 14) the configuration object file (block 10), and the appropriate run-time support libraries. The linker command file (block 12) also arranges the object code in accordance with the prototype processor system's memory layout as specified in response to prompter 4.

In order for the ICS to function it is necessary that the basics of various prototype processor systems be included as working parameters. These parameters include information such as maximum memory size and configurations, interrupt procedures, etc. For purposes of the following discussion examples will be included of an ICS system which can interface a Pascal language program with any of the following microprocessors: 8086, 8086/8087, 8088, and 8088/8087.

Before the Pascal object code (Block 14) produced by a compiler can be executed, it must be supplemented with configuration object code (Block 10) produced by ICS. If the designer/programmer already knows the details of the prototype processor system he can create and process an ICS integration source file (block 6) even before he begins programming in Pascal. In fact, the ICS integration source file (Block 6) provides a concise, human-readable description of the hardware/software interface, and can be used as a design document.

It is likely, however, that the software will be developed in parallel with the hardware, and that parts of the program will be tested before the entire program develops. In testing parts of the program in emulation mode, the default ICS configuration object file and linker command file can be used. As interrupt processing routines are added to the program and it is moved to the prototype processor system, the ICS integration source file (Block 6) will be created and modified to match the program's changing environment.

When the ICS system is initiated, prompter 4 queries the user for information as to the prototype processor system configuration. Table 1 shows a typical set of prompters, user responses, and a comment field forming a typical integration source file (Block 6).

The ICS Prompter (Block 4) is an interactive program that creates the ICS integration source file. Prompter 4 asks questions about the prototype processor system, object files and interrupt configuration, and builds the file according to the responses.

When ICS is invoked processor 4 introduces itself and displays a menu of options to choose from. In the "question mode", prompter 4 begins asking questions and building the integration source file. Immediately after ICS is invoked, processor 4 first creates a default integration source file as shown in Table 3 and then modifies it according to the designer's specifications.

As discussed above, the integration source file is applied to processor 8 to generate the linker command file (Block 12) and the configuration object file (Block 10). Table 2 is a typical link command file which would be generated from the integration source file of Table 1.

When ICS processor 8 finishes reading the integration source file (Block 8), it invokes the assembler of the host computer to create the configuration (Block 10) and listing from the integration source file (Block 6) is automatically deleted after the assembler finishes.

If ICS processor 8 finds any errors in the ICS integration source file, it does not produce the linker command file or configuration object file and the listing shows only the ICS directives and the associated error messages.

The ICS configuration object code (Block 10) performs the following tasks:

—sets up interrupt vectors;
—sets up interrupt service routines;
—sets up a reset routine; and
—sets up program initialization routines.

ICS produces two types of code to carry out these tasks:

—Interrupt handling code—Interrupt vectors and the register save/restore routines that may be used in interrupt handling.

—Initialization and reset code—Code that initializes the segment registers (CS, DS, SS, ES), the stack pointer, the heap pointers, the BP register, the 8087, and the floating point status word variable. This code may also include a routine called COPYVECTORS that copies the interrupt vectors from ROM to RAM, and also a RESET vector which creates the reset code (a JMPS instruction) at location FFFFO.

Most of the code in an ICS configuration object file may be used for interrupt handling. A total of 256 interrupt types, are possible and each type may have a different interrupt service routine. Each interrupt type may need to save and restore the registers of the 8086 and 8087. The actual code generated by ICS depends upon the ICS integration source file choices for each interrupt type.

Two sections are generated simultaneously for interrupt servicing code; a section to contain the interrput vectors (ICS.VROM) and a section to contain the executable code (ICS.INSTR). (The "RESUME" assembler directive acts as a switch, first declaring some code in one section and then some code in the other section.)

Each interrupt service routine has a portion of code devoted to its handling. The assembly code for handling each interrupt type follows a general pattern.

The general pattern shown below represents a portion of the ICS listing. Uppercase words are either actual assemble directives or instructions. Lowercase words are a description of the function of the assembler code and represent one or more lines from the listing.

```
;VECTORRESUME ICS.INSTRINTSERVE,m
VECTOR$ SET $
                save same registers and preserve traceback
                call interrupt servicing routine
                restore registers
                IRET
                RESUME ICS.VROM
                create interrupt vector in memory
```

In the first listing line, ;VECTOR is the ICS directive that declares the interrupt service routine and the type(s) handled by that routine. INTSERVE represents the routine that services the interrupt type number "m".

The SET directive causes VECTOR$ to become a pointer. VECTOR$ points to the beginning of the interrupt handling code for this interrupt type.

The "save some registers and preserve traceback" code performs two functions: saves selected registers and allows the runtime error checking routines to trace the source of runtime errors.

The "call interrupt service routine" code calls the routine that services the interrupt type.

The "restore registers' code restores those registers that were saved. (Refer to Table 4 for a list of ICS subroutines that are used to save and restore registers.)

The IRET instruction returns control to the interrupted routine.

The RESUME ICS.VROM directive causes any lines of code that follow the directive to be placed in the ICS.VROM section.

Next, ("create interrupt vector in memory") a vector is created in memory pointing to the beginning of the interrupt handling code. This is done with two assembler directives (an example is shown in the sample ICS listing). The first defines the location where the interrupt vector is to be placed. The second creates the value of the interrupt vector in that location.

The sample ICS listing, later in this section, shows the particular code produced for each of the interrupt types in a sample ICS source file.

Table 4 lists the subroutines that may be included in the ICS object code.

Interrupt types specified with the INTERRUPTS__ TYPES__ USED directive, but not mentioned in a VECTOR directive, are referred to as undefined interrupt types. The FAULT__ NOTIFICATION directive is used to specify the interupt handling routine for these undefined types. The RESTART__ LABEL directive generates interrupt vectors for these undefined interrupts. See the sample ICS listing, shown later in this section, for an example.

If the INTERRUPT__CONFIGURATION is RAM, the interrupt vectors must be created (in CONSTANTS__ROM) and then transferred to the interrupt vector area (ICS.VRAM) at runtime. The FAULT__NOTIFICATION directive reserves the appropriate areas in ICS.VRAM for the

interrupt vectors and sets up the code to transfer the vectors there during program initialization.

The table created in ICS.VROM has the form shown in Table 5.

The program initialization code is created by the RESTART__LABEL directive starting at the location PASCAL__BEGIN. The initialization code is executed at runtime and performs the following tasks:

—sets the DS, SS, and ES registers to the base of the data segment (DATABASEQQ);
—sets the stack pointer value to STKBASEQQ minus DATABASEQQ;
—initializes the heap pointers at HEAPBASEQQ;
—if necessary, calls COPYVECTORS$ to copy the interrupt vectors from ICS.VROM to ICS.VRAM;
—if necessary, initializes the 8087 by doing an FINIT and initializing the control word;
—clears the global variable floating point status word;
—clears the BP register so that the traceback routine knows that this is the main program; and
—jumps to the main program's entry point MAINQQ.

The reset code is a JMPS instruction, placed at location FFFFOH, pointing to the initialization code. After the microprocessor is reset, this JMPS instruction causes the initialization code to be executed.

The RESTART__LABEL directive also includes the code for any interrupt-related subroutines that are needed (as listed in Table 4.)

If the configuration object code generated by ICS does not fit the prototype processor system application the code ICS produces may be modified.

When ICS is invoked, if the — l (listing) or — o (object code) option if specified, and the ICS integration source file contains no errors, ICS generates temporary assembly language source file from the ICS directives. If the o-option is included, ICS invokes the 8086 assembler to create the object file (filename.io) and assembler listing (filename.il). The assembler invocation that ICS uses looks something like this:

```
asm filename.io filename.il  /lib/8086/ics.mc  /tmp/XXXXXTnnnnTnn
      |             |             |                  |
   object        listing      ICS macro        temporary assembly
   file          file         definitions      language source
                 (if requested)                file
```

The input to the assembler consists of two files, which are processed as if they were concatenated into a single file. The first file, /lib/8086/ics.mc, consists mostly of assembler macro definitions that are used by the second file. The second file is the temporary assembly language source file that ICS has just created. After the assembler finishes, the temporary file is automatically deleted.

If the -o option is omitted, the temporary assembly language source file is saved as the listing file (filename.il). If the designer would like to modify the code that ICS produces, he can edit this source file or create a modified version of the ICS macro file. However, modifying the code that ICS produces may cause your program to be linked, loaded or executed incorrectly.

For example, suppose the designer wants to change the code that saves and restores 8086 registers. This code is found in ics.mc, in macros SAV-86-INLINE$, RES-86-INLINE$, and SAVE-RESTORE-86$. He creates his own copy of ics.mc and modifies those three macros:

```
cp/lib/8086/ics.mc mymacros [Make a copy of the ICS macros.]

ed mymacros                 [Modify his copy.]
```

Each time he invokes ICS, he can create the assembly language source file but skip the assembly step:

```
ics — l myprog. is          [Generate assembly language
                            source file in myprog.il.]
```

Then he assembles the source file using his own macros:

```
asm myprog.io myprog.list   [Generate the object file
mymacros myprog.il          myprog.io and listing file
                            myprog.list]
```

This assembly command line effectively concatenates his macro definitions (*mymacros*) with the ICS produced *myprog.il* and then assembles this file.

Table 6 lists the assembly language macros in the ics.ms file and their functions.

The ICS listing file produced by ICS from a hypothetical ICS integration source file is now shown. By examining the ICS listing file the ICS configuration object code that is linked into the executable load module that runs on the prototype processor system can be understood.

ICS creates a temporary file from the ICS integration source file. This file is assembled and produces the ICS configuration object file and the assembly listing called the ICS listing file. Table 7 is an actual listing produced by ICS.

The following is an explanation of the listing of Table 7.

(During assembly, the assembler uses macro definitions from the file /lib/8086/ics.mc. The macro definitions and their line numbers from the ics.mc file are not shown in the listing.)

Lines 1–187 are from ics.mc; the remaining lines in the listing are the ICS directives and the assembly language statements they create. (Lines 11–186 are the macro definitions from ics.mc, which are not listed.)

Lines 4–9 set up section ICS.INSTR and the standard global symbols, and associate the value DATABASEQQ with the DS register.

Line 202 sets up section ICS.VROM, which will contain the interrupt vectors.

The VECTOR directive on line 207 generates lines 208–238. Lines 211–235 save the 8086 registers in-line, call the interrupt service routine (called NMI), restore the registers, and return from the interrupt.

Lines 236–238 create an interrupt vector at address 0008 that points to the beginning of the register save code (line 212). This spot was marked by the "VECTOR$ SET $" directive on line 210.

Lines 237 and 238 (ORG and WORD assembler directives) are typical of lines that create an interrupt vector. ORG defines the location of the vector. This code is placed in the interrupt vector table at address 8H (for interrupt type 2). WORD creates the interrupt vector. The first word created is VECTOR$-CODEBASEQQ. This word is the offset from the CS register which points to the code that saves registers (starting at location 0, lines 212). The second word created, BITS(CODEBASEQQ,4,16) contains the top 16 bits of CODEBASEQQ (which has 20 bits). These 16 bits will be placed in the CS register when the interrupt occurs.

The VECTOR directive on line 239 generates lines 240–249. This directive code is similar to the previous code (lines 211–235), except that the 8086 registers are not saved and restored in-line. Instead, they are saved and restored using the sub-routines SAV.86$ and RES.86$, the code for which will be generated by the RESTART___LABEL directive.

The VECTOR directive invocation on line 250 generates the lines 251–262. Since the SAVE-FLOATING___POINT and EXCEPT___ for directives direct ICS to save the floating point registers for interrupt types 32 and 34, lines 255–257 save the FPSWQQ variable and the 8087 status on the stack before the call to the interrupt service routine, and restore them afterward.

The VECTOR directive on line 263 generates lines 264–275. This directive is essentially the same as the previous directive.

The VECTOR directive on line 276 generates lines 277–282. Because of the OWN___CODE parameter, no register save and restore code is generated. The four interrupt vectors that are generated all vector directly to TIMER (lines 279–282).

All remaining code is generated in response to the RESTART___LABEL directive (line 283).

The FAULT___NOTIFICATION directive (back at line 206) specified that undefined interrupts are to be handled by the interrupt service routine called BAD___INTERRUPT. Since interrupt types 33 and 35 are undefined (not listed in any VECTOR directives), lines 284–291 generate the code necessary to save the 8086 registers, call BAD___INTERRUPT, and restore the registers. Lines 292–296 generate interrupt vectors to this code for types 33 and 35.

Lines 298–352 generate the code for the 8086 and 8087 register save and restore routines: SAV.86$, RES.86$, SAV.87$, and RES.87$.

The program initialization code starts at label PASCAL___ BEGIN (line 356). Lines 357–361 initialize DS, SS, and SP. Lines 362–366 initialize the heap pointers. Lines 368–374 initialize the 8087. Lines 377–381 initialize ES and BP. Line 384 jumps to the main Pascal program.

Lines 386–391 set up the RESET vector FFFF0.

Table 8 shows the interrupt type, the name of the interrupt service routine for that type, the ICS code that saves and restores registers after an interrupt, and the line numbers of the code in the ICS listing.

To create an executable load file, the linker is involved, specifying only the linker command file that ICS has created and the load file to be created.

Table 9 is a sample listing of the prompter (block 4) routine and table 10 is a sample listing of the processor (block 8) routine.

Referring now to Fig. 2 there is shown a computer terminal 40 and a host computer 30. Terminal 40 includes a CRT 42, keyboard 43, processor 41, memory 44 and timing and control 45. In addition, terminal 40 includes I/O 46 to enable two way communication between the terminal and host computer 30.

Host computer 30 includes CPU 32 which operates under the control of timing and control

36. In addition, host computer 30 includes memory 34 and I/O 38. I/O 38 is the communications link between host computer 30 and the peripherals, i.e. terminal 40, printer/plotter 39 and PROM programmer 37.

In operation, the prompter and processor routines, like those included in Tables 9 and 10, are initially stored in memory 34. When the designer, via keyboard 43 of terminal 40, initiates the ICS system the prompter routine is called up by CPU 32 from memory 34. The individual prompts are then transmitted to the CRT of terminal 40 via I/Os 38 and 46. The designer enters his responses via keyboard 43 which are then transmitted back to CPU 32 where the prompter routine formulates the integration source file (Fig. 1, block 6).

Next the processor routine is called-up by CPU 32 to convert the integration source to the linker command file and the configuration object file as discussed above.

When these steps are completed, the machine independent program to be conditioned for use on the prototype processor system is loaded into host computer 30, together with the standard run-time support library for the language in which that program is written. If host computer 30 includes a complier for the language, the machine independent program can be entered in either source or object form. In the above example the language of the desired program was to be Pascal.

With the four above-identified files, the linker routine of host computer 30, under the control of the linker command file, links the configuration object file, the Pascal object file, and selected as necessary routines of the Pascal run-time library to create the executable load module (Fig. 1, block 20). The executable load module includes, in machine language, the information and locations in the ROMs of the prototype processor system for that system to operate as per the program included in the Pascal object file.

With the proper peripherals, the host computer 30 can list the executable load module, via printer/plotter 39, or program test PROMs, via PROM programmer 37.

CLAIMS

1. A method of integrating a high level language program together with the hardware limitations of a selected prototype processor, the method comprising the steps of:

a. interactively preparing a source file containing software, hardware and interrupt configuration specifications of the selected prototype processor in response to designer inputs; and

b. processing the source file of step a. to generate a linker command file and a configuration object file.

2. A method as in claim 1 further comprising the steps of:

c. linking the high level language program with the configuration object file under the control of the linker command file to generate a load module executable by the prototype processor.

3. A method as in claim 1 or 2 wherein step a. includes the steps of:

d. prompting the designer to input software, and prototype processor hardware and interrupt specifications; and

e. generating a source file containing those specifications.

4. A method as in any preceding claim further comprising the step of:

f. linking the high level language program, the configuration object file and selected routines from a run-time library for the high level language under the control of the linker command file to generate a load module executable by the prototype processor.

5. A method as in any preceding claim wherein said configuration object file of step b. includes interrupt vectors, interrupt service routines, a reset routine and a program initialization routine for the prototype processor.

6. A method as in any of claims 1 to 4 wherein said linker command file of step b. includes routines for linking the high level language program, the configuration object file appropriate run-time support library routines.

7. An integration control system for integrating a high level language program together with the hardware limitations of a selected prototype processor comprising:

means for interactively preparing a source file containing software, hardware and interrupt configuration specifications of the selected prototype processor in response to designer inputs; and

means for processing the source file to generate a linker command file and a configuration object file.

8. A system as in claim 7 further comprising:

means for linking the high level language program with the configuration object file under the control of the linker command file to generate a load module executable by the prototype processor.

9. A system as in claim 7 or 8 wherein the interactively preparing means includes:

means for prompting the designer to input software, and prototype processor hardware and interrupt specifications; and

means for generating a source file containing those specifications.

10. A system as in claim 7 further comprising:

means for linking the high level language program, the configuration object file and selected routines from a run-time library for the high level language under the control of the linker command file to generate a load module executable by the prototype processor.

11. A system as in any of claims 7 to 10 wherein said configuration object file includes interrupt vectors, interrupt service routines, a reset routine and a program initialization routine for the prototype processor.

12. A system as in any of claims 7 to 11 where said linker command file includes routines for linking the high level language program, the configuration object file and appropriate run-time support library routines.

13. A method of integrating a high level language program substantially as herein described with reference to and as illustrated in the accompanying tables and drawings.

14. An integration control system substantially as herein described with reference to and as illustrated in the accompanying tables and drawings.

Printed for Her Majesty's Stationery Office by Burgess & Son (Abingdon) Ltd.—1984.
Published at The Patent Office, 25 Southampton Buildings, London, WC2A 1AY, from which copies may be obtained.